碩士學位論文

PLinda 시스템과 임베디드 시스템을 이용한 움직임 감지 시스템의 구현 및 성능평가

2004 年

漢城大學校 一般大學院

컴퓨터 工學科

컴퓨터 工學 專攻

洪 錫 用

碩士學位論文
指導敎授 朴 永 煥

PLinda 시스템과 임베디드 시스템을 이용한 움직임 감지 시스템의 구현 및 성능평가

Implementation of the Movement Detection Systems
Using PLinda and Embedded System
and Performance Analysis of Them

위 論文을 컴퓨터工學 碩士學位論文으로 提出함

2004年 12月 日

漢城大學校 一般大學院

컴퓨터 工學科

컴퓨터 工學 專攻

洪 錫 用

碩士學位論文

指導教授 朴 永 煥

PLinda 시스템과 임베디드 시스템을 이용한 움직임 감지 시스템의 구현 및 성능평가

Implementation of the Movement Detection Systems Using PLinda and Embedded System and Performance Analysis of Them

2004年 12月 日

漢城大學校 一般大學院

컴퓨터 工學科

컴퓨터 工學 專攻

洪 錫 用

洪錫用의 工學 碩士學位論文을 인정함

2004年 12月 日

심사위원장 이 민 석 (인)

심사위원 정 인 환 (인)

심사위원 박 영 환 (인)

목 차

1. 서 론 ········· 1
1.1 배 경 ········· 1
1.2 관련 연구 ········· 3
1.2.1 부분 윤곽선 추출 기법 ········· 3
1.2.2 PLinda System ········· 4
1.3 연구 목표 ········· 8

2. 본 론 ········· 9
2.1 Cell 단위 부분 윤곽선 추출 기법 ········· 9
2.1.1 부분 윤곽선 추출 기법의 한계 ········· 9
2.1.2 Cell 단위 부분 윤곽선 추출 기법 ········· 12
2.1.2.1 가상 공간(virtual zone)의 설정 ········· 13
2.1.2.2 Cell 단위 부분 윤곽선 추출 ········· 14
2.2 Embedded System을 이용한 움직임 감지 시스템의 구현 ········· 17
2.2.1 하드웨어 ········· 17
2.2.2 움직임 감지 시스템의 구성 ········· 20
2.3 PLinda를 이용한 움직임 감지 시스템의 병렬화 ········· 24
2.3.1 PLinda를 이용한 움직임 감지 시스템의 구조 ········· 24
2.3.2 시스템의 각 세부 사항 ········· 25
2.3.2.1 Camera Server ········· 25
2.3.2.2 PLinda System ········· 27
2.3.2.3 PLinda Movement Detection System의 구현 ········· 29
2.4 성능 분석 ········· 31
2.4.1 가상 공간의 크기와 처리 시간과의 관계에 대한 평가 ········· 31
2.4.2 임베디드 시스템을 이용한 움직임 감지 시스템의 성능 평가 ········· 33
2.4.3 PLinda를 이용한 병렬 처리 움직임 감지 시스템의 성능 평가 ········· 35

3. 결 론 ········· 39

참고문헌 ········· 41
부 록 ········· 43
1. Master Program(main.plc) ········· 43
2. Worker Program(clientmain.plc) ········· 48
ABSTRACT ········· 50

표 목 차

표 1 Capture Sub-routine ··· 20
표 2 Movement detection routine의 3가지 인터페이스 ··· 21
표 3 PLinda 움직임 감지 시스템의 하드웨어 사양 ··· 35

그림 목차

그림 1 부분 윤곽선 수행 과정 3
그림 2 Linda Model 4
그림 3 PLinda Transaction 6
그림 4 PLinda System 6
그림 5 Tuple-space Groups 7
그림 6 PLinda 기본 연산 7
그림 7 움직임 감지 대상 구역 9
그림 8 감시 대상 구역을 벗어난 상황 10
그림 9 불필요한 부분 윤곽선 추출 10
그림 10 가상 공간의 설정 13
그림 11 부분 윤곽선 추출 절차 14
그림 12 소프트웨어 그레이 이미지 변환기 15
그림 13 아날로그 CCD 카메라 17
그림 14 Frame Grabber 18
그림 15 Embedded Board 18
그림 16 움직임 감지 시스템 하드웨어 19
그림 17 Embedded System 상의 Movement Detection System 구조 20
그림 18 전체 영상에 대한 좌표 계산 21
그림 19 부분 윤곽선 추출 절차의 순서 21
그림 20 가상 공간과 셀의 크기 입력 22
그림 21 움직임 감지 시스템의 실행 23
그림 22 PLinda를 이용하여 병렬화한 움직임 감지 시스템의 구조 24
그림 23 RBS(Request Based Server) 방식의 Camera Server 25
그림 24 Camera Server와 Client간 메시지 송수신 절차 26
그림 25 PLinda System 27
그림 26 PLinda 3.1 for Redhat Linux 8 28
그림 27 One Cycle of Movement Detection 29
그림 28 PLinda 소스의 컴파일 과정 30
그림 29 가상 공간의 크기와 처리 시간의 관계 31
그림 30 변화된 셀의 개수에 따른 처리 시간의 변화 32
그림 31 임베디드 시스템을 이용한 움직임 감지 성능 평가 1 33
그림 32 임베디드 시스템을 사용한 움직임 감지 성능 평가 2 34
그림 33 참여 노드의 개수에 따른 성능 분석 1 36
그림 34 참여 노드의 개수에 따른 성능 분석 2 37

그림 35 참여 노드의 개수에 따른 성능 분석 3 ·· 38

1. 서 론

1.1 배 경

눈부시게 발전하는 과학 기술에 힘입어 인간은 보다 더 편리한 생활을 영위할 수 있게 되었고 앞으로도 더욱 편한 생활을 이루게 될 것이다. 이러하듯 기술의 발달에 힘입어 인간의 역할 비중이 줄어들고 있고 대신 인간의 삶은 더욱 자유로워지고 있다.

인간의 역할을 기술에 위임하기 위해서 기술은 인간 이상의 수행 능력을 발휘해야만 한다. 그래야만 인간이 마음놓고 다른 일을 할 수 있다. 즉, 기술의 효과가 커야 함을 의미한다. 하지만 아직까지는 인간의 모든 역할을 대신할 정도로 기술이 따라오지 못한 것 같다.

많은 분야에서 이런 무인 기술을 적용하려고 시도하고 있고, 특정 분야에서는 어느 정도 성공을 거두고 있다. 비전 시스템도 이 중 하나일 것이다. 사람의 눈을 대신하여 카메라를 연결시킨 시스템이 특정 사물을 인지하거나 구역을 감시할 수 있다. 이는 항상 인간의 주의를 요하는 분야에서 인간을 대신할 수 있고 오히려 인간을 능가하는 감시 역량을 발휘할 수 있다.

기술의 발달에 힘입어 카메라는 단순히 영상 정보만을 입력받아서 데이터를 전달하는 역할 뿐 아니라 입력 영상 데이터에서 특정 정보를 추출하거나 환경에 반응할 수 있는 시스템으로 존재할 수 있는 가능성이 열렸다. 이를 가능하게 해주는 기술 중 중요한 것 하나는 임베디드 시스템일 것이다. 카메라에 내장하여 본래 카메라의 역할에 지능을 더 추가할 수 있는 여지가 생긴 것이다. 임베디드 시스템과 비전 시스템의 만남은 카메라로 하여금 무한한 응용력을 갖게 한다. 이렇게 탄생한 지능형 카메라 시스템은 무인 감시나 무인 추적 등을 가능하게 하여 인간이 해야 하는 단순하고 위험한 역할을 대리할 수 있을 것이다.

이 시스템은 카메라에서 입력되는 연속적인 이미지 데이터를 처리해야 한다. 이미지 데이터는 그 크기가 크기 때문에 큰 메모리 공간을 차지하고 많은 CPU 처리 시간을 요구한다. 또 이미지 데이터의 처리는 보통 여러 단계의 처리 과정을 포함하기 때문에 고사양의 자원을 필요로 한다[1, 2]. 따라서 사양이 좋지 않은 임베디드 시스템을 사용하기 위해서는 최적화되고 효율적인 알고리즘을 사용해야만 하고 불필요한 처리는 최소화시켜야 한다.

만약 처리를 전담하는 시스템의 사양이 낮은 경우에는 네트워크에 연결된 다른 유휴 자원을 사용할 수 있다. 최근의 네트워크는 수 십 Mbps에서 수 Gbps까지 높은 대역폭으로 구성할 수 있다. 만약 이런 고속의 네트워크에 연결된 유휴 컴퓨터를 이용할 수 있다면 처리 효율을 더욱 재고할 수 있을 것이다.

보통 여러 기관의 내부 네트워크로 연결된 컴퓨터들은 그 사용 시간이 일정하며 대부분의 시간에 사용되지 않기 때문에 프로세스 자원을 낭비하게 된다. 이런 유휴 자원을 사용하여 많은 컴퓨팅 자원을 필요로 하는 곳에 이용하려는 연구는 이미 오래 전부터 계속되어 왔고 Grid 시스템 같은 일부 시스템들에 대해 현재 활발한 연구가 진행되고 있다[3, 4].

1.2 관련 연구

1.2.1 부분 윤곽선 추출 기법

연속적인 이미지 사이에서 움직임을 감지하는 방법은 일반적으로 두 순차적인 이미지의 차이를 계산해서 변화를 감지하는 것이다[5, 6]. 영상의 차이를 계산하는 것은 데이터의 저장을 위한 큰 메모리 공간과 많은 CPU 사이클 등 적지 않은 자원을 소모한다. 부분 윤곽선 추출 기법은 이런 이미지 데이터의 처리에 효율적인 방법을 제공한다.

한 이미지에서 움직이는 물체는 국부적이다. 즉, 이미지 전역에 걸쳐 변화가 발생하는 것이 아니라 이미지의 일부분에 변화가 생기는 경우가 대부분이라는 것이다. 부분 윤곽선 추출 기법은 이러한 사실에 착안하여 영상 속에서 변화된 부분을 찾아 그 부분에 대해서만 움직임 감지 절차의 전 과정을 수행하게 된다[1, 2, 5]. 이는 이미지 전체에 대하여 변화된 부분이 작을수록 더 좋은 처리 속도를 얻을 수 있게 한다.

부분적인 변화 정보는 시간 미분 결과를 사용하여 얻는다. 이미지에 대하여 시간 미분을 하여 변화 전후의 국부적인 정보를 원 영상에서 추출하여 윤곽선을 발견하고, 그 부분에 대해서만 시간 미분 결과에 곱셈 연산을 수행하면 시간 미분법 중 하나인 Murray 기법과 동일한 결과를 얻을 수 있다(그림 1)[1].

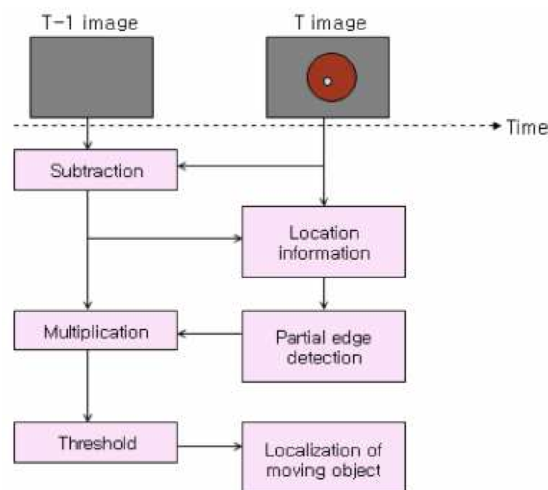

그림 1 부분 윤곽선 수행 과정

1.2.2 PLinda System

PLinda 시스템은 Linda 시스템에 fault tolerant 기능을 추가한 병렬 시스템이다. Linda는 Yale University의 David Gelernter와 그 팀에 의해 제안된 병렬 프로그램의 한 모델이다[7]. Linda는 프로세스에서 시간적, 공간적 문제를 분리해 냄으로써 순차적 프로그램에 들어가는 정도의 노력으로 병렬 프로그램을 개발할 수 있게 하였다[8, 9, 10]. Linda 모델은 tuple과 tuple-space, 그리고 4개의 tuple-space 연산으로 구성되어 있다(그림 2).

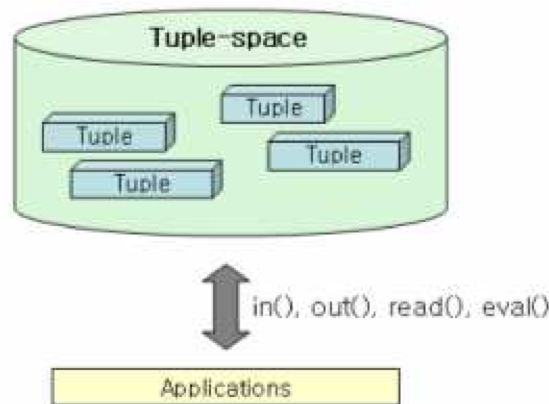

그림 2 Linda Model

Tuple은 각 tuple 고유의 키(key) 필드와 데이터의 집합으로 구성되어 있다. 그리고 tuple-space는 tuple들로 구성되어 있는 일종의 공유 연상 기억장치(shared associative memory)로, 이 안에서 각 tuple은 메모리의 주소가 아니라 자신의 고유 키를 통하여 검색될 수 있다.

Tuple-space에 대한 4가지 연산에는 in(), out(), read(), eval()이 있다. out()은 out(keys, datas)의 형식으로 tuple-space에 한 개의 tuple을 삽입하게 된다. 이 때 keys는 그 tuple의 고유한 식별자가 되고 datas는 tuple의 데이터 값이 된다. 이렇게 삽입하는 과정에서 이 tuple을 읽기 위해 in()이나 read()를 실행시킨 프로세스가 있는지 검사하여 이 들을 깨운 후(wake-up) 프로세스 대기 큐(ready queue)에 추가한다. eval()은 tuple을 tuple-space에 추가시킨다는 점에서 out()과 비슷하나 eval(key, function(e), true)과 같이 사용되어 function(e)을 실행하여 그 결과가 사실이면 결과를 tuple로 생성하여 tuple-space에 삽입시킨다. eval()은 Linda

모델에서 새로운 프로세스를 생성시키는 유일한 수단이다. in()과 read()는 tuple-space에서 원하는 키를 갖은 tuple을 읽기 위한 수단이지만 in()은 해당 tuple을 tuple-space로부터 삭제시킨다는 점에서 read()와 다르다. 두 연산을 수행할 시에 원하는 tuple이 없는 경우 그 연산을 호출한 프로세스는 수면상태(sleep state)로 들어가게 된다.

Linda 모델에서는 프로세스간 통신(inter-process communication)을 수행하는데 있어서 데이터가 메모리 주소가 아닌 키를 매개로 이루어지기 때문에 병렬 프로그램을 개발할 경우 데이터의 공간적인 위치를 고려할 필요가 없다. 또 해당 tuple이 존재하지 않을 경우 프로세스를 수면상태로 만들고 해당 tuple이 입력될 경우 깨어나기 때문에 프로세스간 동기화(inter-process synchronization) 문제를 고려하지 않고 쉽게 프로그램을 개발할 수 있다.

Linda는 병렬 프로그램을 간단한 방법으로 작성 가능하게 하고, 복잡한 문제들은 모두 감추어 효율적인 병렬 시스템을 구축하게 하지만, 병렬 컴퓨터에서 중요한 고려 사항 중 하나인 fault-tolerant 기능이 존재하지 않는다. 오랜 시간에 걸쳐 계산해온 데이터가 시스템의 장애나 오류로 손실되면 다시 처음부터 계산을 해야 하기 때문이다. 이런 문제점을 해결하기 위한 Linda의 확장판 중 하나가 PLinda(Persistent Linda)이다[11, 12].

PLinda는 트랜잭션(transaction)과 체크포인트(checkpoint) 개념을 사용하여 tuple-space와 프로세스를 보호한다. 트랜잭션은 프로그램을 트랜잭션 단위로 구분하여 처리하는 것을 의미하며, 트랜잭션 내의 연산이 모두 끝날 때 tuple-space에 데이터를 저장하기 때문에 프로세스 오류로 인한 잘못된 데이터의 저장을 방지하고 오류로 인해 사라진 작업을 트랜잭션 번호를 추적하여 복구할 수 있다(그림 3).

체크포인트는 트랜잭션 단위로 처리된 정보를 tuple-space에 기록하는 것이다. 복구 시 체크포인트 된 정보를 추적하여 복귀가 진행된다.

PLinda 시스템에는 서버가 존재하며 이 서버는 tuple-space를 생성하고 각 PE(Processing Element)에 데몬을 호출한다. 각 데몬은 자신의 PE가 idle한 상태에 있을 경우 tuple-space에 접근하여 작업을 할당받게 된다(그림 4). 만약 PE가 idle하지 않은 경우 PLinda 서버는 프로세스를 다른 idle한 PE로 이동시켜 수행하게 된다.

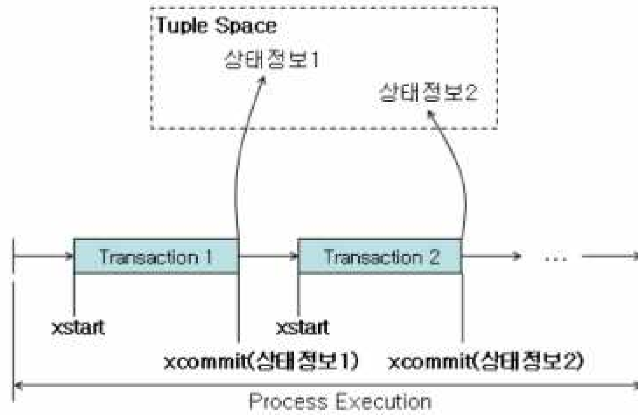

그림 3 PLinda Transaction

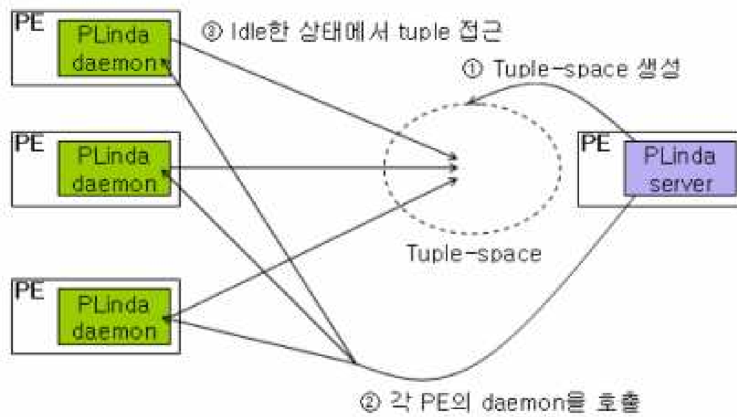

그림 4 PLinda System

PLinda는 같은 패턴을 갖는 다른 응용의 tuple들이 서로 충돌하지 않게 하기 위하여 각 응용간 그룹(group)을 두어 문제점을 해결했다. 각각의 tuple 그룹은 create_group()로 생성되고, destroy_group()에 의해 해제된다. create_group()은 생성된 그룹의 gid를 반환하고 이 후로는 이 gid를 통해서 tuple에 접근 가능하다(그림 5).

PLinda 시스템은 Linda의 fault-tolerant한 확장이므로 Linda의 기본 연산을 제공한다. pl_proc_eval()은 새로운 프로세스를 생성하기 위해 사용한다. 이렇게 생성된 프로세스를 불러내기 위해 사용하는 연산은 pl_arg_rdp()이다. Master 프로세스에서 pl_proc_eval()을 호출하고, worker 프로세스에서 pl_arg_rdp()를 호출하여 프로세스를 실행하게 하는 것이다. pl_out(), pl_in(), pl_rd()는 각각 Linda 모델의 out(), in(), read()와

동등하다. 하지만 PLinda에서는 group 단위 별로 tuple-space에 접근하기 때문에 gid를 다음과 같은 방법으로 명시해야 한다(그림 6).

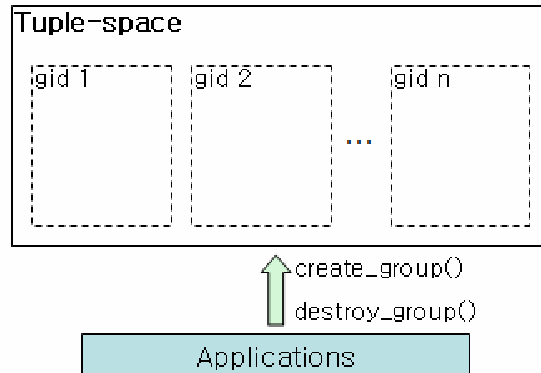

그림 5 Tuple-space Groups

```
pl_out[gid]("Tuple");
pl_in[gid]("Tuple");
pl_rd[gid]("Tuple");
```

그림 6 PLinda 기본 연산

PLinda는 fault-tolerant 기능을 포함하므로 Linda에 비해 몇 가지 연산을 더 제공한다. xstart/xcommit()은 프로그램 상에서 트랜잭션을 구분하는 역할을 한다. 즉 한 트랜잭션은 xstart로 시작해서 xcommit()으로 끝나게 된다. xcommit()은 트랜잭션의 상태를 저장하기 위한 tuple을 매개로 사용한다.

한 프로세스가 미쳐 특정 트랜잭션을 commit하지 못하면 PLinda 시스템은 그 프로세스를 처음부터 다시 시작하는 것이 아니라 바로 전에 commit한 트랜잭션 이후부터 수행을 하게 된다. xrecover()는 성공한 마지막 트랜잭션에 관련된 tuple을 읽어서 마지막 프로세싱을 시작해야 할 지점과 상태를 복구하게 된다.

1.3 연구 목표

카메라를 이용한 응용 중 실제 사회에서 가장 빈번하게 사용할 가능성이 높은 분야 중 하나는 무인 감시 시스템일 것이다. 항상 사람이 카메라 영상을 감시하는 일은 단조롭고 귀찮은 것이다. 더군다나 한시라도 주의를 기울이지 않으면 안되기 때문에 지능형 카메라 시스템의 사용은 더욱 필요할 것이다. 앞에서도 언급했듯이 카메라로 입력된 이미지를 처리하는 작업은 상당히 많은 메모리 공간과 프로세스 시간을 필요로 한다. 게다가 이런 무인 감시 시스템을 임베디드 보드에서 수행한다면 더 많은 제약이 따르게 된다. 그렇기 때문에 움직임 감지 절차의 효율성이 전체 시스템 성능의 큰 부분에 영향을 미치게 된다.

본 연구에서는 움직임 감지 시스템의 성능 향상을 위하여 효율적인 처리 방법을 제시하고 실제 시스템을 구현하고자 한다. 높은 처리 효율을 갖는 시스템을 개발하여 임베디드 보드에 이식하고 이렇게 개발된 시스템의 성능을 측정하려 한다. 또 단체 내부의 가용 자원을 활용하고자 병렬처리 시스템 중 하나인 PLinda를 이용하여 무인 움직임 감지 시스템을 병렬화하고 그 활용 가능성을 확인하기 위해 성능을 평가한다.

본 논문에서 포함하고 있는 내용은 다음과 같다.

1. 효율적인 움직임 감지 방법의 제시
2. 임베디드 보드에 움직임 감지 시스템 구현
3. PLinda를 이용한 움직임 감지 시스템의 병렬화 가능성
4. 구현된 시스템들의 성능 평가

본 논문의 본론 부는 위와 같은 순서로 구성되어 있으며 각 연구 주제마다 세부적인 사항을 설명할 것이다.

2. 본 론

2.1 Cell 단위 부분 윤곽선 추출 기법

2.1.1 부분 윤곽선 추출 기법의 한계

기존 부분 윤곽선 추출 기법은 이미지 상 변화된 부분의 정보를 이용하여 변화가 있는 경우에만 움직임 감지 전 처리 과정을 수행하였다. 따라서 움직임이 없는 경우에는 처리 성능을 크게 향상시킬 수 있었다[1, 5]. 하지만 대부분의 이미지에서 움직임을 감시해야 할 필요가 있는 부분은 전체 중 일부에 지나지 않는다. 다음 이미지를 보자.

그림 7 움직임 감지 대상 구역

위의 이미지는 건물 출입구에 카메라를 설치하고 출입구에 사람이 지나가는가를 감시하는 예이다. 이 이미지에서 사각형이 그려진 출입구 부분에 대해서만 감시를 하면 된다. 만약 부분 윤곽선 추출 기법을 사용하게 된다면 변화된 부분에 대한 정보를 얻기 위해서 한번은 시간 간격을 둔 두 이미지의 차를 계산해야 한다. 이는 변화가 없는 경우에 불필요한 처리 오버헤드를 요구한다. 물론 카메라를 출입구만 보이게 설치하면 되지만 일반적으로 카메라의 위치는 고정되어 있고 설치할 수 있는 장소도 제약이 있기 때문에 쉽지 않은 문제이다. 또, 그렇게 고정을 해놓으면 시스템 가동 시에 동적으로 관심지역을 선택할 수 없게 된다.

감시 대상이 되는 지역에 변화가 없지만 관심이 없는 부분에 변화가 생

기는 경우를 고려해 보자. 다음은 그런 경우의 예이다.

그림 8 감시 대상 구역을 벗어난 상황

그림 8을 보면 출입구가 아닌 다른 부분에 변화가 생겼다. 이런 경우 부분 윤곽선 추출 기법은 변화가 발생했기 때문에 부분 윤곽선 처리 과정 전부를 수행하고 변화된 부분의 좌표를 얻을 것이다. 이 좌표가 관심 지역에 포함되어 있지 않기 때문에 추가 조치는 취하지 않겠지만 결국 처리할 필요가 없는 모든 과정을 다 수행하게 된다(그림 9).

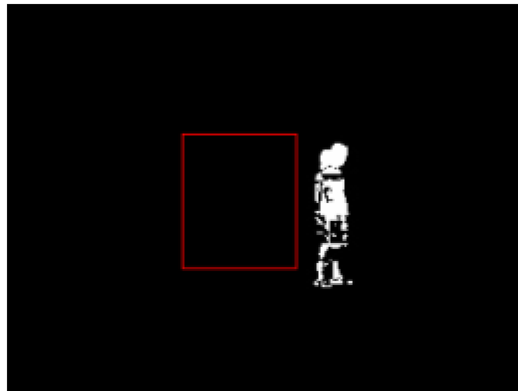

그림 9 불필요한 부분 윤곽선 추출

부분 윤곽선 추출 기법을 움직임 감지 시스템에 적용할 경우 또 다른 문제가 존재한다. 만약 움직인 물체가 두 개 이상일 경우 각 물체를 구분하는 일은 쉽지 않다. 어디까지가 한 물체에 속하는지 찾아내는 알고리즘을 생각하기 어려운데다 적용한다 하더라도 구현이 복잡하고 역시 큰 오버헤

드가 발생할 수 있기 때문이다[13, 14, 15, 16].

따라서 부분 윤곽선 추출 기법을 움직임 감지 시스템에 적용할 경우 상당히 큰 비효율성을 포함하게 되고 이를 임베디드 시스템에 구현할 경우 자원의 부족이나 처리 시간의 증가 등의 문제가 발생할 수 있다. 따라서 보다 효율적인 방법의 연구가 필요하다.

2.1.2 Cell 단위 부분 윤곽선 추출 기법

본 연구에서 제안하는 방법은 영상을 여러 개의 cell로 나누어 처리하는 부분 윤곽선 추출 기법이다. 본 기법은 다음과 같은 두 개의 부분으로 이루어진다.

1. 가상 공간(virtual zone)의 설정
2. Cell 단위 부분 윤곽선 추출

2.1.2.1 가상 공간(virtual zone)의 설정

우선 전체 이미지에서 관심을 갖는 부분을 선택하고 그 부분을 가상 공간으로 설정한다. 앞으로 모든 움직임 감지 처리는 이 가상 공간 내의 부분에 대해서만 적용하게 된다.

가상 공간을 설정하면 가상 공간 내부를 cell 단위로 분할한다. 움직임 감지 대상이 되는 물체의 크기는 물체에 따라 다르며, 또한 카메라와의 거리에 따라 그 크기가 변화하기 때문에 이런 것을 고려하여 한 물체를 충분히 포함할 정도로 cell의 크기를 지정해야 한다. 이렇게 cell을 지정하면 부분 윤곽선 추출 과정에서 한 cell에 대해 한 물체만 파악하면 되기 때문에 간단한 물체 추출 알고리즘을 적용할 수 있다.

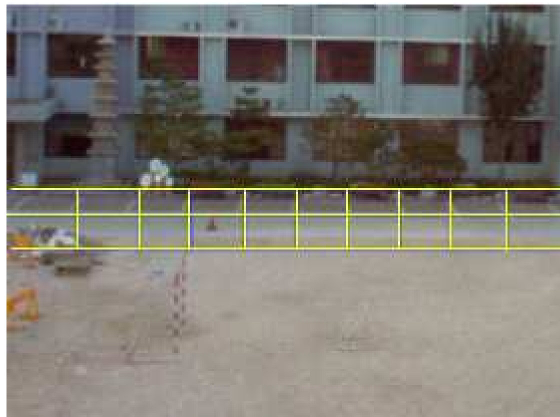

그림 10 가상 공간의 설정

그림 10은 무인 주차 시스템을 위해 주차 구역에 가상 공간을 설정하고 자동차 한 대를 포함할 크기 정도로 cell을 분할한 경우이다.

2.1.2.2 Cell 단위 부분 윤곽선 추출

가상 공간이 설정되고 cell로 분할되면 각 cell의 이미지에 대해 부분 윤곽선 추출 기법을 적용한다. 부분 윤곽선 추출 방법은 기존의 순서대로 수행을 하게 된다(그림 11).

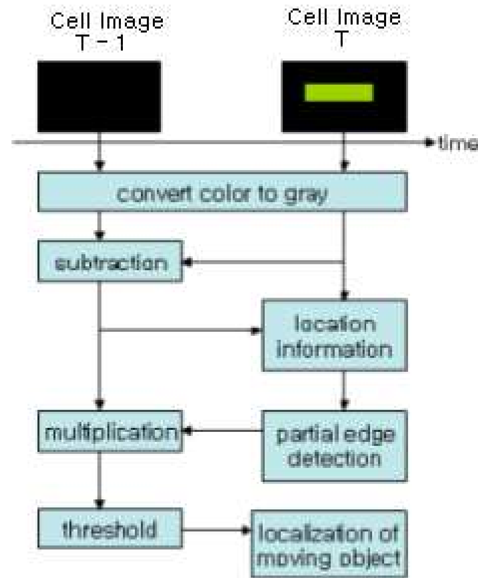

그림 11 부분 윤곽선 추출 절차

전체 이미지를 카메라로부터 얻을 경우 아날로그 CCD 카메라를 사용하려면 아날로그 신호를 디지털 신호로 변환하기 위한 프레임 그래버(frame grabber)를 사용해야 한다. 보통의 경우 이 프레임 그래버는 비트맵 형식의 컬러 이미지를 전송하지만 설정을 달리함으로써 그레이 이미지(gray image)를 얻을 수 있다. 만약 사용하는 프레임 그래버가 그레이 이미지를 지원하고, 구현하는 시스템에서 컬러 이미지를 필요로 하지 않을 경우 이런 하드웨어 수준의 그레이 이미지를 사용하면 메모리 공간과 처리 속도를 더욱 높일 수 있다.

만약 하드웨어 수준에서 그레이 이미지를 지원하지 않지만 그레이 이미지를 사용해도 무방한 시스템에서는 소프트웨어 변환기를 사용하는 것이 좋다. 다음은 3bytes 비트맵 이미지를 1byte 그레이 이미지로 변환해 주는 변환기의 코드이다(그림 12).

```
void color_to_gray(char *c_image, unsigned int *g_image) {
        unsigned int *img_pixel, sum, temp, index;
        img_pixel = (unsigned int *)c_image;
        for(index = 0;index < img_size_integer;index++){
                sum = *img_pixel & 0x000000FF;
                sum += (*img_pixel & 0x0000FF00) >> 8;
                sum += (*img_pixel & 0x00FF0000) >> 16;
                temp = (int)(sum / 3);
                g_image[index] = temp;
                sum = (*img_pixel & 0xFF000000) >> 24;
                img_pixel++;
                sum += *img_pixel & 0x000000FF;
                sum += (*img_pixel & 0x0000FF00) >> 8;
                temp = (int)(sum / 3);
                g_image[index] = g_image[index] | (temp << 8);
                sum = (*img_pixel & 0x00FF0000) >> 16;
                sum += (*img_pixel & 0xFF000000) >> 24;
                img_pixel++;
                sum += *img_pixel & 0x000000FF;
                temp = (int)(sum / 3);
                g_image[index]=g_image[index] | (temp << 16);
                sum = (*img_pixel & 0x0000FF00) >> 8;
                sum += (*img_pixel & 0x00FF0000) >> 16;
                sum += (*img_pixel & 0xFF000000) >> 24;
                temp = (int)(sum / 3);
                g_image[index]=g_image[index] | (temp << 24);
                img_pixel++;
        }
}
```

그림 12 소프트웨어 그레이 이미지 변환기

소프트웨어 변환기를 사용하면 컬러 이미지를 그레이 이미지로 변환하는 오버헤드가 발생하지만 부분 윤곽선 처리 전 과정에서 저장해야 하는 메모리 공간과 프로세스 시간이 줄기 때문에 전체적으로 더 좋은 성능을 얻을 수 있다.

2.2 Embedded System을 이용한 움직임 감지 시스템의 구현

2.2.1 하드웨어

본 시스템에 사용된 카메라는 컬러 이미지를 입력받는 CCD 방식의 것을 사용하였다. CCD 카메라는 제어 기능 같은 특별한 기능이 없는 단순한 것일 경우 저렴한 가격에 구입할 수 있는 장비이다. 디지털 방식의 카메라를 이용할 수도 있고, 이럴 경우 프레임 그래버가 추가로 필요 없지만 대개의 경우 감시용 카메라는 아날로그 방식의 CCD 카메라를 사용하기 때문에 본 연구에서는 아날로그 CCD 카메라를 이용하였다(그림 13).

그림 13 아날로그 CCD 카메라

아날로그 카메라를 사용하였기 때문에 컴퓨터에서 이미지를 획득하기 위해서는 아날로그 이미지를 디지털로 변환시켜주는 프레임 그래버를 사용하여야 한다. 프레임 그래버는 그 기능에 따라 상당히 가격 차이가 심하기 때문에 원하는 기능을 고려하여 적당한 제품을 선택해야 한다. 본 연구에서는 벨기에 Euresys사의 Picolo Pro 2라는 프레임 그래버를 사용하였다(그림 14). 이 장치는 NTSC나 PAL 같은 대표적인 영상 신호를 처리하며 실시간 영상 획득을 지원하고 RGB32, RGB24, Y8 등 다양한 이미지 포맷을 하드웨어 수준에서 지원한다. 또, Linux를 지원하여 디바이스 드라이버나 간편한 사용을 위한 API를 제공하는 이점이 있다. 본 연구에서는

Linux를 기반으로 시스템을 구축하기 때문에 이 제품을 선택하였다.

그림 14 Frame Grabber

임베디드 보드는 가격이 비교적 저렴한 싱글보드(single board) 컴퓨터를 이용하였다(그림 15).

그림 15 Embedded Board

프로세서는 i386 호환의 제품을 사용하였고 750MHz의 클럭 주파수를 사용한다. 메모리는 128Mbytes의 SDRAM을 장착하고 무엇보다도 프레임

그래버가 PCI 인터페이스를 갖고 있기 때문에 PCI 슬롯을 하나 이상 포함해야 한다.

운영체제로는 Redhat Linux 8을 사용하였다. 우선 Linux는 쉽게 구할 수 있으며 임베디드 보드를 위해 다운사이징(down sizing)이 가능하기 때문이다. 그리고 프레임 그래버가 Redhat Linux 8을 지원하기 때문에 본 연구에서는 상기의 OS를 선택하였다.

그림 16 움직임 감지 시스템 하드웨어

2.2.2 움직임 감지 시스템의 구성

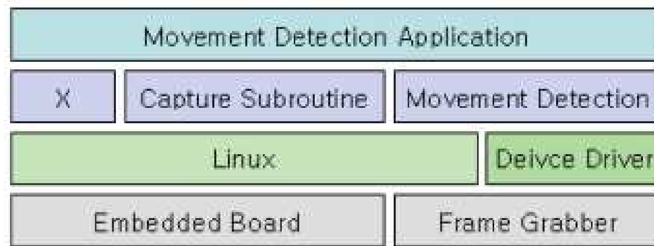

그림 17 Embedded System 상의 Movement Detection System 구조

움직임 감지 시스템 응용 프로그램은 그림 17과 같이 구성하였다. 실제 운영상에서는 화면에 이미지를 표시할 필요가 없지만 테스트용으로 이미지를 화면에 표시하게 하였고 X 라이브러리를 사용하였다. 프레임 그래버에서 640 × 480 픽셀 크기의 이미지를 얻도록 설정하였기 때문에 윈도우의 크기를 640 × 480 픽셀로 설정하였다.

Capture sub-routine은 프레임 그래버에 접근하기 위한 API들을 사용하여 디바이스 드라이버를 활성화/비활성화 하는 루틴과 프레임 단위로 이미지를 얻는 루틴을 작성하였다(표 1).

함수	기능
int initCapture(void)	디바이스 드라이버 활성화
void closeCapture(void)	디바이스 드라이버 비활성화
unsigned char *getCaptureImage(void)	한 프레임을 캡쳐

표 1 Capture Sub-routine

Movement detection routine은 앞에서 설명한 부분 윤곽선 추출 기법의 전 과정을 포함하고 있으며 두 이미지를 입력으로 받아서 움직인 물체를 포함하는 최소 사각형의 top left 좌표와 bottom right 좌표 그리고 화면 표시를 위하여 발견된 물체의 윤곽선 이미지를 반환한다. 반환된 윤곽선 이미지는 테스트를 위한 화면 출력에 포함하여 실제 움직임이 감지된 상태를 표시하기 위해 사용한다.

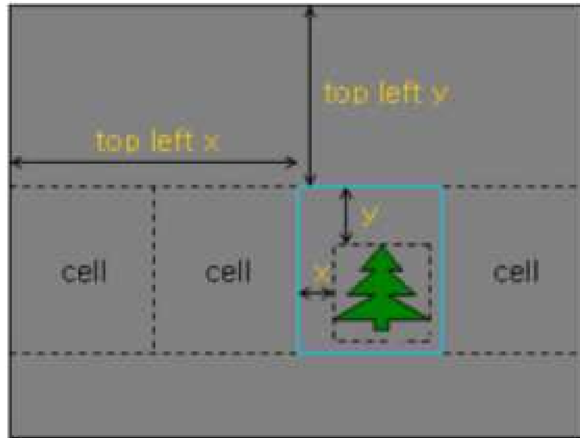

그림 18 전체 영상에 대한 좌표 계산

Movement detection routine은 크게 3가지 사용자 인터페이스를 제공하도록 구성하였으며 다음과 같다.

함수	**기능**
void init_movement(long x, long y)	환경 변수 및 메모리 생성
void close_movement(void)	메모리 해제
rect treat_movement (unsigned char *, unsigned char *)	실제 두 개의 이미지를 받아서 부분 윤곽선 추출 과정을 수행

표 2 Movement detection routine의 3가지 인터페이스

treat_movement()는 다음과 같은 함수를 수행하여 실제 부분 윤곽선 추출 절차를 수행한다(그림 19).

```
soustraction(이미지1, 이미지2);
이미지2 = edge_filter_sobel(이미지2);
이미지1 = mul_image(이미지1, 이미지2, THRESHOLD);
locate(이미지1);
```

그림 19 부분 윤곽선 추출 절차의 순서

제안한 cell 단위 움직임 감지 기법은 사용자가 이미지 전체에서 임의로 가상 공간을 설정할 수 있어야 한다. 따라서 응용 프로그램 실행 전에 가

상 공간의 좌표와 cell의 개수를 입력받아야 한다(그림 20).

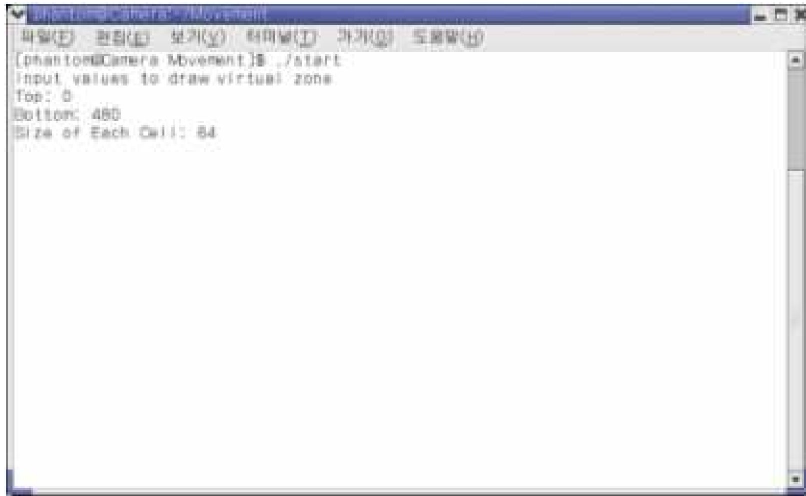

그림 20 가상 공간과 셀의 크기 입력

위의 그림은 처음 움직임 감지 시스템을 작동할 때 나타나는 가상 공간과 셀의 크기를 입력하는 예이다. 위의 예에서 가상 공간은 (0, 0)부터 (640, 480)로 설정되었고, 각 셀의 넓이는 64 픽셀로 설정되었다. 이는 가상 공간을 정적으로 고정시키는 것이고, 필요하다면 프로그램 수행 중에 동적으로 설정할 수 있을 것이다.

이렇게 가상 공간과 각 셀의 위치를 설정하면 그림 21과 같이 움직임 감지 절차를 수행한다. 그림 21은 640 × 480 픽셀의 크기를 갖는 이미지에 대하여 (0, 200)부터 (640, 400)까지 가상 공간을 설정하고, 셀의 크기를 가로 64 픽셀, 세로 100 픽셀로 설정한 것이다. 앞에서도 언급했듯이 X 라이브러리를 사용하여 데모용으로 카메라 화면과 감지된 물체의 윤곽선을 혼합하여 화면에 출력하였다.

부분 윤곽선 추출을 하는 과정에서 많은 임계치(threshold)들이 존재하고 각 임계치로 설정된 값에 따라서 처리 결과가 상당히 달라지기 때문에 카메라 calibration을 해주는 것이 무엇보다도 중요하다.

그림 21 움직임 감지 시스템의 실행

2.3 PLinda를 이용한 움직임 감지 시스템의 병렬화

2.3.1 PLinda를 이용한 움직임 감지 시스템의 구조

Cell 단위 움직임 감지 시스템을 PLinda를 이용하여 병렬화를 하였다. 구현한 시스템의 구조는 다음과 같다.

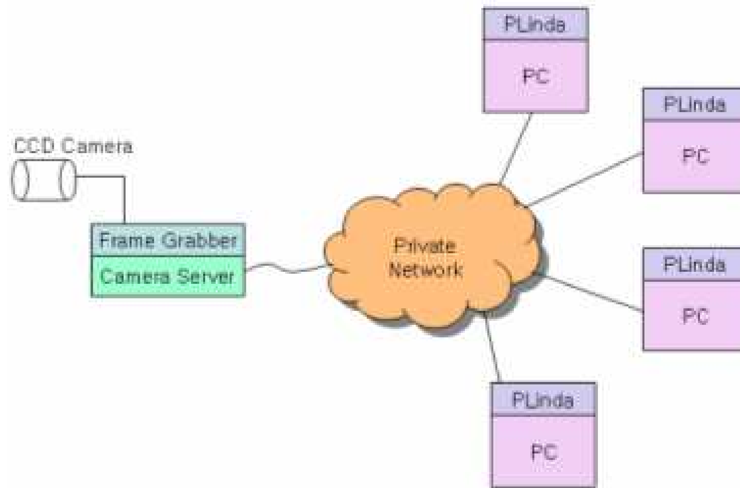

그림 22 PLinda를 이용하여 병렬화한 움직임 감지 시스템의 구조

이미지 획득을 위한 방법으로 임베디드 시스템을 이용한 움직임 감지 시스템에 사용했던 아날로그 CCD 카메라와 Picolo Pro 2 프레임 그래버를 사용하였다. 카메라 서버를 위해 역시 같은 임베디드 보드를 사용했으며 서버는 RBS(Request Based Server)를 구현하였다[17]. 카메라 클라이언트는 정해진 메시지 형식에 따라 TCP 연결로 이미지를 획득할 수 있으며 쓰레드를 기반으로 동작하기 때문에 다중 클라이언트를 지원한다. 병렬 처리를 위하여 여러 대의 PC를 카메라 서버와 함께 사설 네트워크로 구성하였으며 각각의 PC에는 PLinda 시스템을 설치하였다.

2.3.2 시스템의 각 세부 사항

2.3.2.1 Camera Server

카메라 서버는 임베디드 보드를 사용하였으며 100Mbps의 LAN 카드를 내장하고 있다. 서버는 병렬 처리를 위하여 사용한 PC들과 함께 사설 네트워크로 묶었다.

카메라 서버는 다중 클라이언트를 지원한다. RBS(Request Based Server) 방식으로 작성하였는데, 이 방식은 클라이언트로부터 요청이 들어오면 쓰레드를 생성하여 클라이언트와 일대일로 통신하는 방법이다[17]. 따라서 동시에 다중 클라이언트의 요청을 원활하게 처리할 수 있다(그림 23).

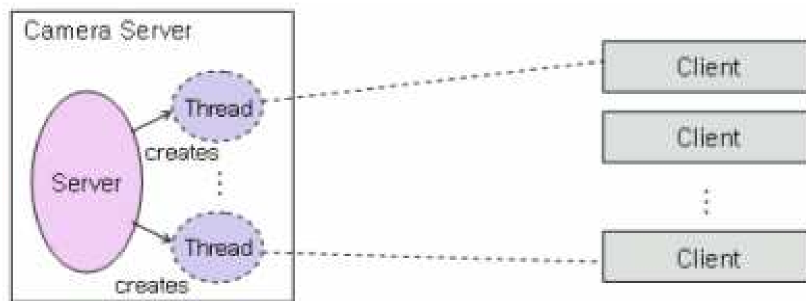

그림 23 RBS(Request Based Server) 방식의 Camera Server

안정적인 이미지 전송을 위하여 서버와 클라이언트간 TCP 연결을 설정한다. TCP는 흐름제어 재전송 메커니즘을 갖고 있기 때문에 UDP 방식보다 더 신뢰성 있는 통신을 가능하게 한다[18]. 카메라 서버는 클라이언트로부터 프레임 요청 메시지를 수신하면 쓰레드를 생성하고 그 이후 진행되는 과정은 전부 생성된 쓰레드가 전담하게 된다.

쓰레드는 우선 REQUEST 메시지를 수신하게 되는데 이 경우 카메라 서버에 접근할 수 있는 ID와 PASSWORD 정보를 포함하고 있다(그림 24). 이때 인증에 실패하면 쓰레드는 클라이언트와의 연결을 종료한다. 인증을 받은 경우에는 Capture 모듈로부터 프레임을 하나 얻어서 클라이언트 측으로 송신하고 접속을 종료하게 된다. 인증을 위한 사용자 ID와 PASSWORD는 현재 코드 상에 정의되어 있지만 추후 확장을 위해 인증

모듈의 구현이 필요하다.

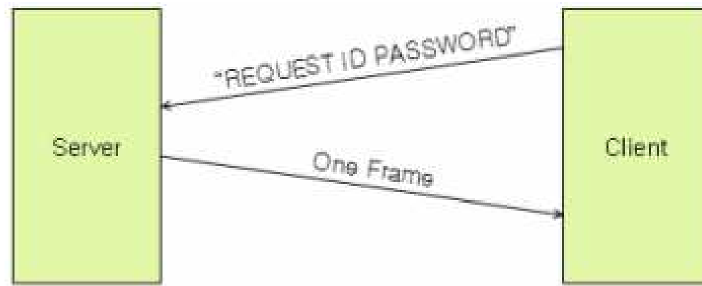

그림 24 Camera Server와 Client간 메시지 송수신 절차

2.3.2.2 PLinda System

PLinda 시스템을 구축하기 위해 범용 PC를 사설 네트워크로 연결하였다(그림 25). 각각의 PC는 heterogeneous한 환경을 위하여 각기 다른 CPU 클럭과 일정하지 않은 크기의 메모리를 장착하고 있으며, 모든 PC는 100Mbps의 LAN 인터페이스 카드를 장착하였다. 사설 네트워크를 위해 100Mbps의 switching hub를 사용하였다.

그림 25 PLinda System

Linux 배포판 중 최근에 많이 사용되는 버전은 Redhat Linux 8/9일 것이다. 모든 시스템은 Redhat Linux 9를 설치하였다.

현재 웹 상에서 구할 수 있는 PLinda의 최신 버전은 PLinda 3.1이다. 하지만 PLinda 3.1은 Redhat Linux 7에서만 작동을 한다. 따라서 Redhat Linux 8이나 9를 사용하기 위해서 PLinda 3.1을 수정해야 했다. 문제의 원인이 되는 것은 크게 두 가지이다.

첫 번째는 Redhat Linux 8/9로 넘어오면서 PLinda에서 사용하는 프로그램들의 경로가 변경되었다는 것이다. PLinda의 소스는 .plc의 확장자가 붙고 plc(PLinda Compiler)에 의해 C 소스 파일로 컴파일된다. 그런데 plc는 Perl 스크립트 언어로 작성되어 있기 때문에 Perl 인터프리터를 사용하게

되는데 이 경로가 Redhat Linux 8 이후부터 변경되었다. 따라서 적당한 위치로 변경을 해주어야 한다.

두 번째로 PLinda 자체의 소스 파일이나 PLinda의 include 파일들이 deprecated된 C/C++ 라이브러리를 사용한다는 점이다. 따라서 deprecated된 모든 함수를 찾아서 그에 동등한 다른 함수로 변경해줘야 한다.

PLinda 시스템을 Redhat Linux 9 상에서 구동시키기 위하여 잘못된 부분은 수정을 하였고 변경한 부분과 설치 방법에 대한 문서도 같이 작성을 하였다. 변경된 PLinda 시스템은 PLinda-Redhat8.tar.gz로 패키징하였으며 문서는 PLinda-Redhat8.doc으로 작성하여 PLinda 패키지의 doc 디렉터리에 함께 포함을 시켰다.

모든 설정을 마치고 PLinda를 실행하면 그림 26과 같이 PLinda 서버가 작동한다.

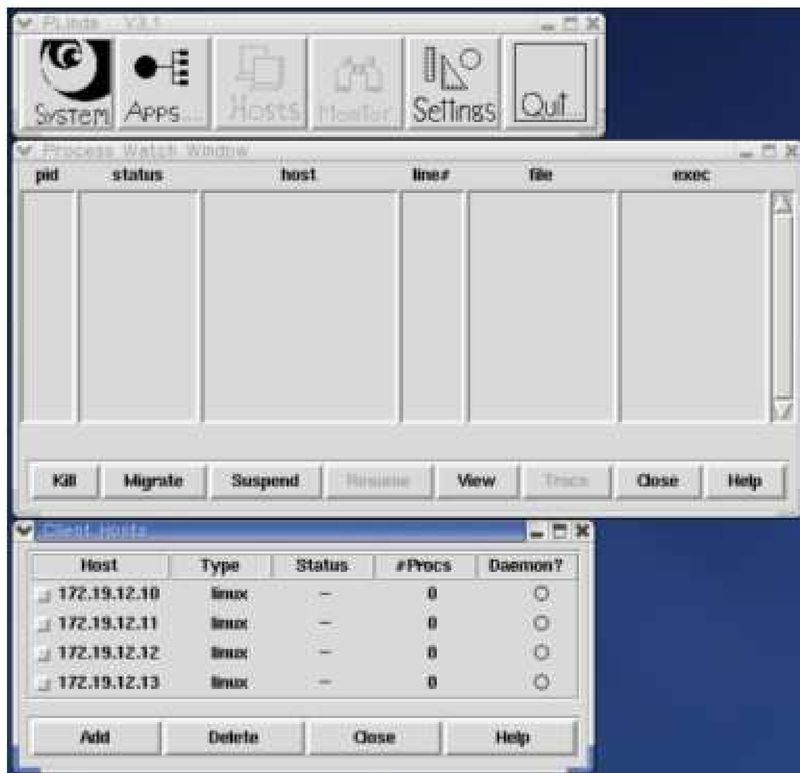

그림 26 PLinda 3.1 for Redhat Linux 8

2.3.2.3 PLinda Movement Detection System의 구현

움직임 감지를 위해 셀 단위로 부분 윤곽선 추출 기법을 적용해야 하기 때문에 매우 자연스럽게 병렬 처리 패러다임의 마스터-워커(master-worker) 구조를 적용할 수 있다. 마스터-워커 패러다임은 말 그대로 마스터가 작업을 분할하여 배포하면 워커들이 작업을 하나씩 할당받아서 처리한 후 다시 마스터에게 결과를 전달하는 구조이다[19].

마스터 프로세스는 우선 카메라 서버에 프레임을 요청하고 해당 프레임을 수신하면 사용자가 설정한 값에 따라 가상 공간과 셀을 나눈다. 또 워커 프로세스를 생성하게 된다. 이 후 분할된 한 셀 이미지를 작업의 단위로 하여 tuple-space에 저장하면 각 워커들은 한 작업씩 할당받아 부분 윤곽선 추출기법을 적용하고 그에 따른 윤곽선 이미지나 움직인 물체의 좌표를 계산하여 다시 tuple-space에 전달한다. 그럼 마스터 프로세스는 이 결과를 수집하고 종합하여 출력하게 된다.

작업을 처리하는 방법은 배리어(barrier) 형식과 유사하다. 즉, 하나의 전체 이미지를 처리하는 것을 한 사이클로 구성하였다(그림 27).

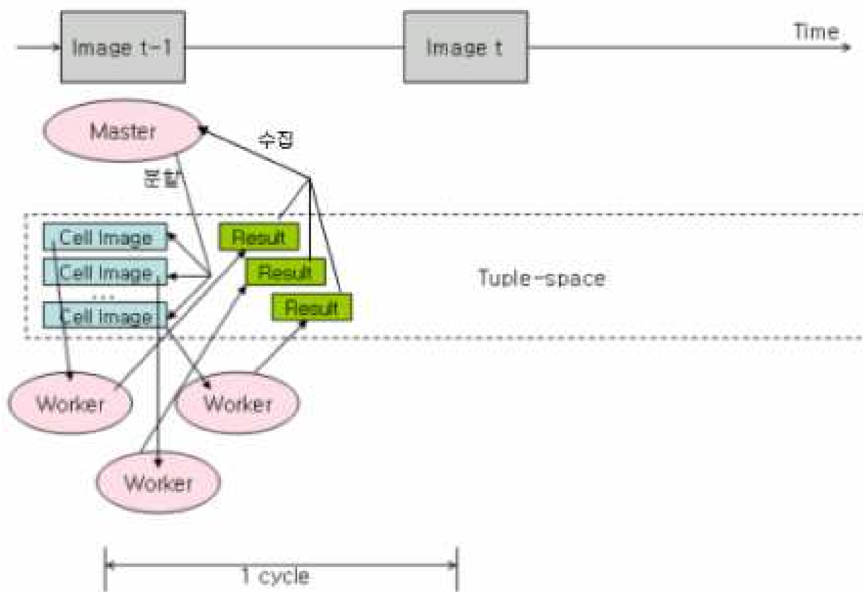

그림 27 One Cycle of Movement Detection

한 프레임을 처리하는 것은 배리어의 한 단계를 수행하는 것과 같으므로

한 프레임을 완전하게 처리할 때까지 다음 프레임을 처리하지 않다가 처리가 끝나는 경우에 다음 프레임을 받아서 같은 작업을 반복하게 된다 [20].

PLinda 응용 프로그램에 대해서 마스터 프로그램은 main.plc, 워커 프로그램은 clientmain.plc로 작성하였다. 소스 코드는 논문의 부록에 첨부하였다. PLinda 소스 코드는 .plc 확장자로 저장한다. 그 다음 plc(PLinda Compiler)는 해당 PLinda 코드를 미리 정의된 적당한 C 코드로 컴파일 하면 .C 확장자의 C 코드가 생성된다. 이렇게 생성된 C 코드는 일반 C 컴파일러를 사용하여 실행 파일로 만들어진다(그림 28).

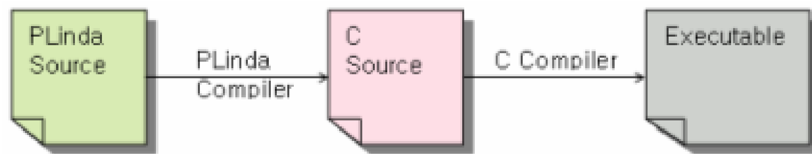

그림 28 PLinda 소스의 컴파일 과정

실행 파일은 마스터를 cameramaster.exe로 워커를 cameraworker로 각각 작성하였다. PLinda 시스템에서 마스터 프로그램은 확장자로 .exe를 붙여야 한다.

2.4 성능 분석

2.4.1 가상 공간의 크기와 처리 시간과의 관계에 대한 평가

가상 공간의 크기가 본 연구에서 제안한 움직임 감지 시스템의 성능에 어떤 영향을 미치는지 알아보기 위해 실험을 하였다. 이 실험은 셀 단위 움직임 감지 시스템을 Windows XP가 설치된 Pentium 3 600MHz CPU의 일반 PC에서 수행을 하였으며, Picolo Pro 2 프레임 그래버를 사용하였다. 이 그래버는 640 × 480 픽셀의 이미지를 초당 최대 38 프레임까지 처리할 수 있다[21]. 메모리는 128Mbytes SDRAM을 사용하였다. 본 실험에서는 두 가지를 평가하였는데 하나는 가상 공간의 크기 변화에 따라 한 프레임을 처리하는 시간이 얼마나 차이가 나는지 이고, 다른 하나는 변화된 셀의 개수에 따라 처리 시간에 얼마나 차이가 있는지 이다.

640 × 480 픽셀의 이미지에서 가상 공간의 크기를 각각 25%, 50%, 75%, 100%로 변경하면서 실험을 하였다(그림 29).

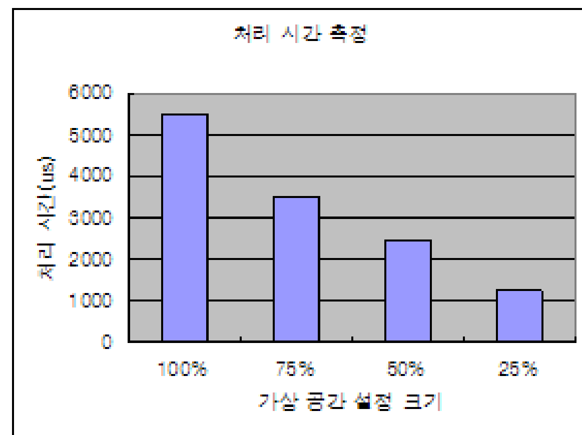

그림 29 가상 공간의 크기와 처리 시간의 관계

실험 결과를 보면 가상 공간의 크기에 따라 처리 시간에 적지 않은 차이가 발생한다는 사실을 알 수 있다. 따라서 불필요한 부분을 보다 많이 제거한다면 움직임 감지 시스템의 효율을 보다 더 재고할 수 있다는 결론을 얻을 수 있다.

두 번째 실험은 변화된 셀의 개수에 따라 처리 시간에 얼마나 차이가 나

는가를 평가하였다. 가상 공간의 크기를 전체 이미지의 50%로 고정하고 20개의 셀을 생성하여 그 중 변화된 셀의 개수를 0%, 25%, 50%, 75%, 100%로 각각 변경하면서 측정하였다(그림 30).

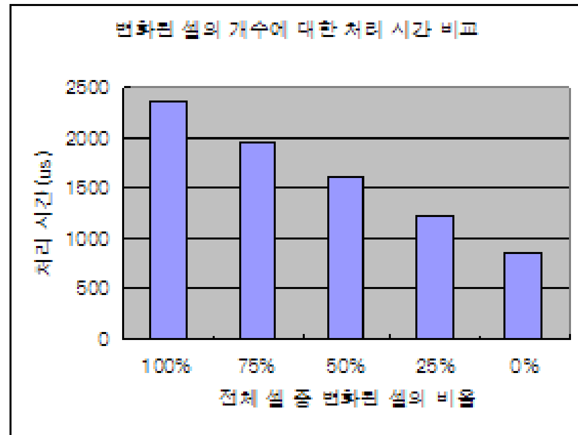

그림 30 변화된 셀의 개수에 따른 처리 시간의 변화

결과를 보면 확실한 사실 하나를 알 수 있다. 부분 윤곽선 추출 기법이 변화 정보를 이용하여 변화된 부분이 없으면 전 움직임 감지 처리 과정을 생략하기 때문에 변화된 셀의 개수에 따라 처리 시간이 비례함을 알 수 있다. 즉, 셀의 크기가 작을수록 변화된 부분이 없을 확률이 크다. 따라서 감지하고자 하는 물체에 대해 그 물체를 포함하는 가장 작은 크기로 셀을 나눈다면 보다 더 좋은 성능을 얻을 수 것이다.

2.4.2 임베디드 시스템을 이용한 움직임 감지 시스템의 성능 평가

성능 분석을 위한 임베디드 시스템용 싱글보드 컴퓨터는 i386 호환의 750MHz 클럭을 갖는 프로세서와 128Mbytes의 SDRAM을 장착하였다. 프레임 그래버는 Euresys사의 Picolo Pro 2로 640 × 480 픽셀의 3Bytes 비트맵 이미지를 초당 최대 38 프레임까지 얻을 수 있는 성능을 갖고 있다. 이 시스템은 다른 어떤 컴퓨터와 연관성 없이 standalone 방식으로 작동하며, 성능 평가의 기준으로 초당 처리되는 평균 프레임의 수를 사용하였다.

첫 번째로 테스트용 영상 출력을 지원하는 버전의 성능을 평가하였다(그림 31).

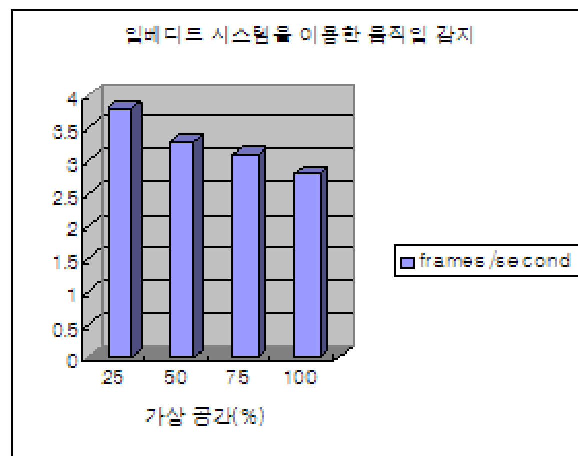

그림 31 임베디드 시스템을 이용한 움직임 감지 성능 평가 1

셀의 개수는 20개로 고정하고 가상 공간의 크기를 각각 전체 영상의 25%, 50%, 75%, 100%로 설정했을 경우 초당 처리한 평균 프레임 수를 측정하였다. 결과를 보면 가상 공간의 크기가 커질수록 초당 처리된 프레임 수가 줄어드는 것을 알 수 있다. 이는 처리해야 할 데이터가 늘어나기 때문으로 명백한 결과이다.

두 번째 평가는 테스트용 영상 출력을 지원하지 않고 오직 움직임 감지 절차만 수행하는 경우이다(그림 32). 만약 본 움직임 감지 시스템을 제품화할 경우 이와 같은 방식으로 시스템을 구축하게 될 것이다.

앞의 경우와 마찬가지로 가상 공간의 크기가 커질수록 평균 처리 속도가

낮아지는 것을 확인할 수 있다. 단지 이 경우는 테스트용 영상 출력을 제외한 경우로 실제 상황에서 이와 같은 방식으로 사용될 것이므로 이 성능이 실제 임베디드 시스템을 사용한 움직임 감지 시스템의 성능에 부합될 것이다. 프레임 그래버가 초당 38 프레임을 지원하고, 움직임 감지 시스템의 처리 시간이 크다는 사실을 고려하면 가상 공간 50%에서 초당 14 프레임을 처리하는 것은 상당히 효율이 좋다고 할 수 있다. 따라서 본 시스템은 임베디드 시스템에 적용하기에 적합하다.

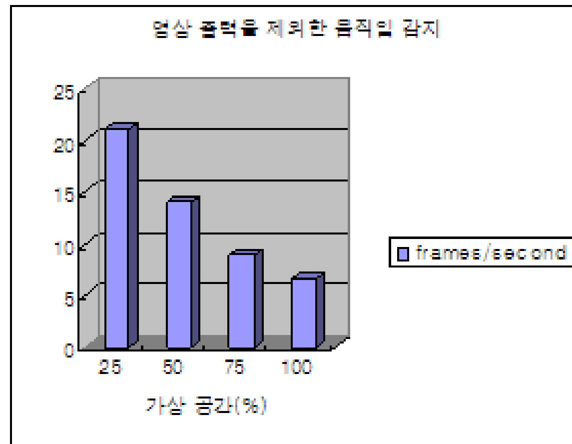

그림 32 임베디드 시스템을 사용한 움직임 감지 성능 평가 2

2.4.3 PLinda를 이용한 병렬 처리 움직임 감지 시스템의 성능 평가

PLinda를 이용한 움직임 감지 시스템을 구성하기 위해 heterogeneous한 PC 8대를 사용하였다. 각 PC의 OS는 모두 Redhat Linux 9이며 주요 하드웨어의 사양은 다음과 같다.

	L1	L2	L3	L4	L5	L6	L7	L8
CPU MHz	P3 700	P2 350	P4 1,800	P2 266	P3 600	P4 2,400	P4 2,400	P4 2,800
RAM MB	128	192	256	192	256	512	512	512
HDD rpm	5,400	5,400	7,200	5,400	7,200	5,400	5,400	7,200
LAN Mbps	100	100	100	100	100	100	100	100
Video RAM MB	64	4	32	16	16	64	64	64

표 3 PLinda 움직임 감지 시스템의 하드웨어 사양

시스템을 구성하는 각 노드의 하드웨어 사양은 표 3과 같이 매우 다양하다. PLinda 응용 프로그램은 일반적으로 master와 worker로 구성되어 있는데 둘 사이의 관계나 작업의 특성, 작업의 크기 등에 따라 그 성능에 적지 않은 차이가 발생한다. 예를 들어 CPU-bounded된 응용 프로그램에서 worker가 master보다 상대적으로 더 많은 연산을 해야 할 경우 성능이 더 좋은 노드에 worker들을 배정하면 더 좋은 결과를 얻을 수 있다[3]. 또 모든 노드의 하드웨어 사양이 동일하다면 노드의 개수에 따라 성능의 차이도 일정하겠지만 본 연구에 사용된 노드들의 하드웨어 사양은 각각 다르기 때문에 결과를 예상하기가 어렵고 경우에 따른 편차도 크다. 따라서 본 연구의 성능 평가 시에 응용 프로그램의 특성을 고려한 최적의 작업 배치에 대한 연구가 선행이 되어야 하지만 그 부분은 고려하지 않았음을 밝힌다.

첫 번째 실험은 가상 공간의 크기를 640 × 480 픽셀 이미지의 25%,

50%, 75%, 100%로 변경하고 각각의 경우에 대해 병렬 처리에 참여하는 노드의 개수를 변화시키면서 처리되는 프레임의 수를 측정하였다. 8대의 PC 모두 사용하였으며 보다 정확한 측정을 위해 master는 L1 PC로 고정하였다. 즉, 모든 경우에 대해 master 프로세스는 L1에서 수행하였다(그림 33).

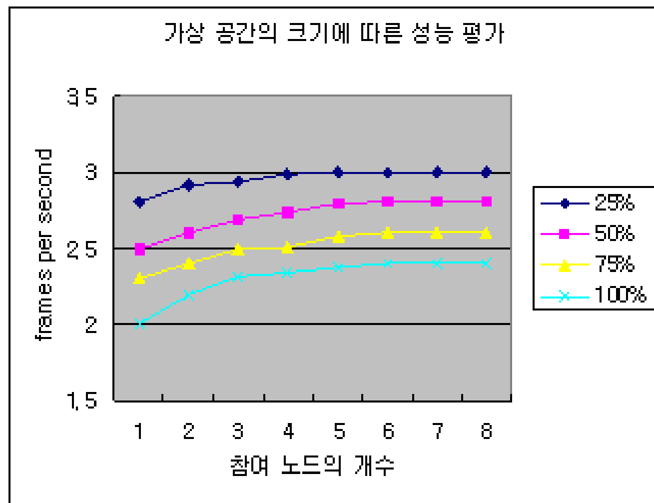

그림 33 참여 노드의 개수에 따른 성능 분석 1

표 3에서 보듯 L1 시스템은 성능이 좋지 않은 편이다. 그리고 성능이 좋은 PC와 그렇지 않은 PC들이 혼재해 있기 때문에 전체적인 시스템의 성능이 비교적 낮다. 하지만 병렬 처리에 참여하는 노드가 늘어날수록 움직임 감지 시스템의 전체적인 성능이 향상됨을 알 수 있다. 그리고 5대 이상의 노드를 참여시켰을 경우 처리 속도에 변화가 없었다. 이는 컴퓨팅 자원이 포화상태로 됐음을 의미한다. 즉 5대 이상의 PC를 PLinda 병렬 시스템에 참여를 시켜도 더 이상 성능의 기대를 하기 힘들다는 의미이다.

다음 실험은 성능이 좋은 Pentium 4 PC 4대(L3, L6, L7, L8)와 성능이 좋지 않은 L4를 사용하여 가상 공간의 크기와 참여 노드의 개수를 변경하면서 성능을 측정하였다(그림 34). 셀의 개수는 20개로 고정하였고 노드를 병렬 처리에 참가시키는 순서는 L4, L6, L8, L3, L7이다. 즉, 한 대의 PC를 사용할 경우 L4, 두 대를 사용할 경우는 L4, L6, 5대를 모두 사용할 경우는 L4, L6, L8, L3, L7을 참여시킨다는 의미이다.

한 대의 노드를 참여시킬 경우 L4의 성능이 매우 좋지 않기 때문에 움직

임 감지 시스템의 성능이 좋지 않음을 알 수 있다. 하지만 노드의 개수가 증가할수록 급격하게 성능이 향상되었다. 4대 이상의 노드를 참여시킨 경우 임베디드 시스템으로 구성한 움직임 감지 시스템의 성능을 추월하였다.

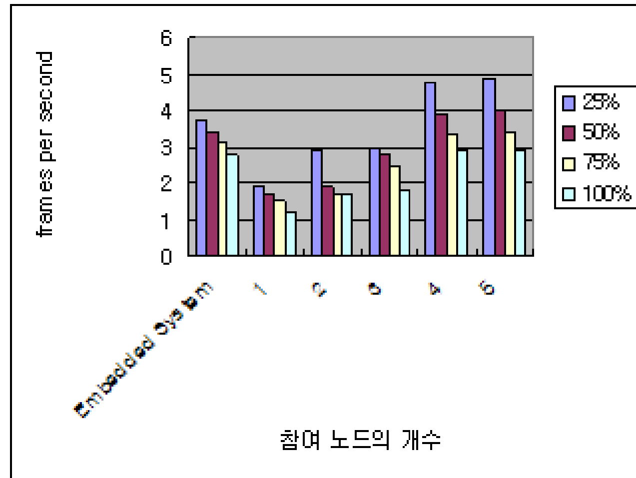

그림 34 참여 노드의 개수에 따른 성능 분석 2

다음 실험은 성능이 좋은 Pentium 4급의 PC 4대(L3, L6, L7, L8)만을 사용하여 성능을 측정하였다(그림 35). 마찬가지로 가상 공간의 크기를 전체 이미지의 25%, 50%, 75%, 100%로 변경하고 병렬 처리에 참여하는 노드의 개수를 하나씩 증가하면서 성능 상의 변화를 관찰하였다. 앞의 실험들과 다른 점은 셀의 개수를 20개가 아닌 4개로 고정시켰다는 점이다. 참여 노드를 추가할 때 L3, L8, L6, L7로 순서를 정하였다.

모든 노드의 성능이 좋기 때문에 모든 경우에 있어서 임베디드 시스템으로 구성한 움직임 감지 시스템보다 성능이 월등함을 알 수 있었다. 그리고 노드의 개수가 3대 이상이 되면서부터 포화 상태가 일어나는 것을 관찰할 수 있다.

당연한 사실이지만 가능하면 좋은 PC들을 병렬 처리에 참여시키는 것이 전체 성능 향상에 좋다. 하지만 무조건 좋은 PC들을 많이 참여시킨다고 성능이 좋아지는 것은 아니다. PLinda 시스템의 중앙 집중적인 메모리 구조로 인한 병목 현상과 네트워크 트래픽, 또 병렬 처리에 참여하는 노드들의 포화 등의 원인 때문이다. 또, 응용 시스템의 특성이나 master-worker의 관계를 파악해서 master 프로세스와 worker 프로세스들을 최적의 노

드에 할당하는 과정도 중요하다[3].

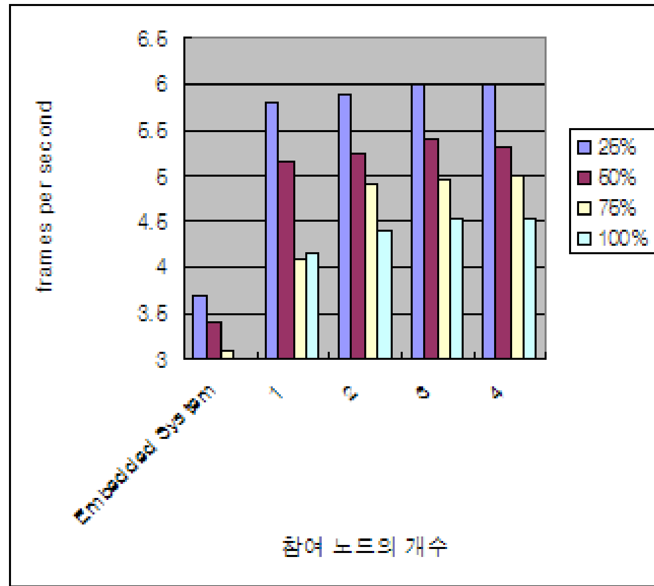

그림 35 참여 노드의 개수에 따른 성능 분석 3

3. 결 론

효율적인 움직임 감지 시스템을 위하여 기존 부분 윤곽선 추출 방식에 가상 공간의 설정 및 셀 단위의 처리 방식을 적용하였다. 가상 공간의 설정은 전체 카메라 이미지에서 관심을 갖는 부분만을 구분하여 전체 처리 성능을 향상시키고자 도입한 기법이고, 셀 단위의 처리는 이미지에서 이동한 물체를 간단하게 알아내기 위한 기법이다. 셀 단위의 처리는 움직인 물체를 쉽게 구분할 수 있도록 할뿐만 아니라 전체 이미지를 보다 작은 크기로 분할하여 움직임이 없는 셀은 부분 윤곽선 추출 과정을 적용하지 않음으로써 전체적인 성능 향상을 이끌어낸다.

카메라와 감지 대상과의 거리에 따라 이미지에서 대상 구역의 크기는 변화한다. 따라서 보다 좋은 결과를 얻으려면 감지 구역을 가능한 작게 나타나도록 카메라의 위치를 조절할 수 있다. 물론 무조건 카메라의 위치를 멀리하는 것은 바람직하지 않을 것이다. 사물의 정확한 식별이나 응용의 목적에 따라 적합한 카메라의 위치를 실험을 통하여 측정해서 선택해야 한다.

제안된 셀 단위 부분 윤곽선 추출 기법을 임베디드 시스템에 적용하여 지능형 카메라 시스템을 구축하였다. 이미지 처리는 적지 않은 메모리와 프로세싱 시간을 필요로 한다. 본 연구에서 제안한 기법은 불필요한 이미지 처리를 최소화하여 보다 적은 자원을 보유한 시스템에서 효율성을 갖도록 한다. 실험을 통하여 알 수 있듯이 가상 공간을 50%로 설정하였을 경우 초당 14 프레임을 처리할 수 있었다. 물론 보다 신속한 처리를 요하는 분야에서는 적합하지 않을지 모르지만 건물 내부의 감시나 무인 불법주차 감시 시스템 등 일반적인 응용 분야에서는 충분히 적합할 것이다.

본 기법은 움직임 감지를 셀 단위로 처리하기 때문에 병렬 처리의 가능성을 내포하고 있다. 즉 셀의 분할과 셀 단위의 처리는 master-worker 패러다임에서 작업의 분할과 분할된 작업을 처리하는 방식에 정확하게 부합한다. 따라서 PLinda 시스템을 이용하여 기관 내부의 가용 자원을 이용할 수 있도록 움직임 감지 시스템을 구축하였으며 적합성을 판단하기 위해 실험을 하였다. 실험을 통하여 셀 단위 움직임 감지 시스템의 병렬 처리 가능성을 확인할 수 있었다. PLinda 시스템의 중앙 집중적인 메모리 구조 때문에 병목 현상이 발생할뿐더러 master 프로세스의 수행을 위한 컴퓨터

의 선택은 응용 프로그램의 특성에 따라 크게 다르기 때문에 실험을 통하여 PLinda 서버와 master 프로세스를 위한 컴퓨터 할당 방법을 찾아야 한다.

참고문헌

[1] 박 영환, 영상 변화 추적 시스템 , 물리탐사, Vol. 2, No. 3, p154-158, 1999

[2] 홍 석용, 박 영환, 가상 공간의 셀 단위 처리를 이용한 움직임 감지 시스템 성능 향상 기법 , 정보과학회 가을 학술발표 논문집, Vol. 31, No. 2, p775-777, 2004

[3] 홍 석용, 박 영환, PLinda에서 master 선택과 성능과의 관계에 관한 연구 , 정보과학회 가을 학술발표 논문집, Vol. 30, No. 1, p388-390, 2003

[4] Viktors Berstis, Fundamentals of Grid Computing , IBM Redbooks Paper, 2002

[5] Murray, D. and Basu, A., Motion Tracking with an Active Camera , IEEE Trans. Patt. Anal. Mach. Intell., 16(5), p449-459, 1994

[6] Li, K., Park, Y. H. and Hou, K. M., Intelligent Camera Dedicated for a Multi-Sensor Perception: Real Time Motion and Object Tracking , International Conference on Advances in Vehicles Control and Safety(AVCS 98), p76-86, 1998

[7] Nicholas Carriero and David Gelernter, LINDA IN CONTEXT , Communications of the ACM, 1989

[8] 박 영환, 병렬 프로그래밍 개념 LINDA의 간편화 , 한국 정보과학회 논문집, 2000

[9] S. Ahuja, N. Carriero, and D. Gelernter, Linda and Friends , Computer, Vol. 19, No. 8, Aug. 1986

[10] David Gelernter, Getting the Job Done , BYTE, p301-308, 1988

[11] Karpjoo Jeong and Dennis Shasha, PLinda 2.0: A Transactional/Checkpointing Approach to Fault Tolerant Linda , Proc. the 13th International Symposium on Reliable Distributed Systems, Oct. 1994

[12] Karpjoo Jeong, Bin Li, Dennis Shasha and Peter Wychoff, PLinda/C++ and Fortran77 Users Guide , March. 17, 1996

[13] Nichol, D. and Fiebig, M., Tracking multiple moving objects by binary object forest segmentation , Image Vision Computing, 9(6), p362-371, 1991

[14] Molineros, J. and Sharma, R., Real-Time Tracking of Multiple Objects Using Fiducials for Augmented Reality , Real-Time Imaging, Vol. 7, No. 6, p495-506, 2001
[15] Iketani, A., Nagai, A., Kuno, Y. and Shirai, Y., Real-Time Surveillance System Detecting Persons in Complex Scenes , Real-Time Imaging, Vol. 7, No. 5, p433-446, 2001
[16] Ziliani, F. and Cavallaro, A., Image Analysis for Video Surveillance Based on Spatial Regularization of a Statistical Model-Based Change Detection , Real-Time Imaging, Vol. 7, No. 5, p389-400, 2001
[17] 정 진국, 낭 종호, 박 성용, 다중 처리기 기반 웹 서버 구조의 실험적 성능 분석 , 정보과학회논문지, 정보통신 제28권 제1호, p22-36, 2001
[18] 강 필용, 신 용태, 실시간 멀티미디어 응용을 위한 IP 멀티캐스트 기반의 QoS 보장 분산 멀티미디어 시스템 , 정보과학회논문지, 정보통신 제28권 제1호, p117-125, 2001
[19] Nicholas Carriero and David Gelernter, How to Write Parallel Programs: A Guide to the Perplexed , ACM Computing Surveys, Vol. 21, No. 3, September 1989
[20] Ananth, G., Anshul, G., George, K. and Vipin, K., Introduction to Parallel Computing , Addison Wesley, 2003
[21] Euresys, Installation Manual , Euresys, 2004

부 록

1. Master Program(main.plc)

```
#include <stdio.h>
#include <plinda.h>
#include <X11/Xlib.h>
#include <sys/time.h>

#include "config.h"
#include "client.h"

Display *display;
Window window;
GC gc;
XImage *ximg;

gid gid_me;
int next_transaction = 0;
int count = 0;

int TOP, MIDDLE, BOTTOM;
int WIDTH;
int NUMBER_OF_WORKERS;

char *frame1, *frame2;
char *image;

int real_main(int argc, char *argv[], char *env[]){
    int index;
    int x, y;
    char *cell_image1, *cell_image2;
    bool recover;
    struct timeval first, last, sub_first, sub_last;

    recover = xrecover(?gid_me(gid), ?NUMBER_OF_WORKERS(int)
              , ?TOP(int), ?MIDDLE(int), ?BOTTOM(int), ?WIDTH(int)
              , ?next_transaction(int));

    display = XOpenDisplay(NULL);
    window = XCreateSimpleWindow(display, DefaultRootWindow(display), 0
```

```
        , 0, IMAGE_WIDTH, IMAGE_HEIGHT, 2, BlackPixel(display, 0)
        , WhitePixel(display, 0));
gc = XCreateGC(display, window, 0L, (XGCValues *)NULL);
XMapWindow(display, window);
XFlush(display);

frame1 = (char *)malloc(IMAGE_WIDTH * IMAGE_HEIGHT
         * IMAGE_DEPTH);
frame2 = (char *)malloc(IMAGE_WIDTH * IMAGE_HEIGHT
         * IMAGE_DEPTH);
memset(frame1, 0x00, IMAGE_WIDTH * IMAGE_HEIGHT
         * IMAGE_DEPTH);
memset(frame2, 0x00, IMAGE_WIDTH * IMAGE_HEIGHT
         * IMAGE_DEPTH);
image = (char *)malloc(IMAGE_WIDTH * IMAGE_HEIGHT * sizeof(int));
memset(image, 0x00, IMAGE_WIDTH * IMAGE_HEIGHT * sizeof(int));

if(next_transaction == 0){
    xstart;
    printf("Input Number of Workers: ");
    scanf("%d", &NUMBER_OF_WORKERS);
    printf("Input Top: ");
    scanf("%d", &TOP);
    printf("Input Bottom: ");
    scanf("%d", &BOTTOM);
    printf("Input Cell Width: ");
    scanf("%d", &WIDTH);
    MIDDLE = TOP + (BOTTOM - TOP) / 2;
    gid_me = create_group("Camera");
    xcommit(gid_me(gid), NUMBER_OF_WORKERS(int), TOP(int)
          , MIDDLE(int), BOTTOM(int), WIDTH(int)
          , ++next_transaction(int));
}
cell_image1 = (char *)malloc(WIDTH * (BOTTOM - MIDDLE)
           * IMAGE_DEPTH);
cell_image2 = (char *)malloc(WIDTH * (BOTTOM - MIDDLE)
           * IMAGE_DEPTH);

if(next_transaction == 1){
    xstart;
    pl_out[gid_me]("CONFIG", WIDTH(int), MIDDLE - TOP(int));
    for(index = 0;index < NUMBER_OF_WORKERS;index++){
        pl_proc_eval("cameraworker", gid_me(gid));
```

```
    }
    xcommit(gid_me(gid), NUMBER_OF_WORKERS(int), TOP(int),
             MIDDLE(int), BOTTOM(int), WIDTH(int)
             , ++next_transaction(int));
}
getFrame((unsigned char *)frame1);

while(1){
    gettimeofday(&first, NULL);
    if(next_transaction == 2){
        getFrame((unsigned char *)frame2);
        xstart;
        for(index = 0;index < IMAGE_WIDTH / WIDTH * 2;index++){
            int col;
            for(col = 0;col < BOTTOM - MIDDLE;col++){
                if(index < IMAGE_WIDTH / WIDTH){
                    memcpy(&cell_image1[col * WIDTH * IMAGE_DEPTH],
                    &frame1[IMAGE_WIDTH * IMAGE_DEPTH * (TOP + col)
                    + WIDTH * IMAGE_DEPTH * index], WIDTH
                     * IMAGE_DEPTH);
                    memcpy(&cell_image2[col * WIDTH * IMAGE_DEPTH],
                    &frame2[IMAGE_WIDTH * IMAGE_DEPTH * (TOP + col)
                    + WIDTH * IMAGE_DEPTH * index], WIDTH
                    * IMAGE_DEPTH);
                 }
                 else{
                     memcpy(&cell_image1[col * WIDTH * IMAGE_DEPTH],
                     &frame1[IMAGE_WIDTH * IMAGE_DEPTH * (MIDDLE
                     + col) + WIDTH * IMAGE_DEPTH * index], WIDTH
                     * IMAGE_DEPTH);
                     memcpy(&cell_image2[col * WIDTH * IMAGE_DEPTH],
                     &frame2[IMAGE_WIDTH * IMAGE_DEPTH * (MIDDLE
                     + col) + WIDTH * IMAGE_DEPTH * index], WIDTH
                     * IMAGE_DEPTH);
                 }
             }
             pl_out[gid_me]("JOB", index(int), cell_image1(char): WIDTH
             * (BOTTOM - MIDDLE), cell_image2(char): WIDTH *
              (BOTTOM - MIDDLE));
         }
         xcommit(gid_me(gid), NUMBER_OF_WORKERS(int), TOP(int),
         MIDDLE(int), BOTTOM(int), WIDTH(int), ++next_transaction(int));
     }
```

```
if(next_transaction == 3){
    xstart;
    for(index = 0;index < IMAGE_WIDTH / WIDTH * 2;index++){
        int col;
        pl_in[gid_me]("RESULT", index(int), ?cell_image1(char): WIDTH
        * (BOTTOM - MIDDLE));
        if(index < IMAGE_WIDTH / WIDTH){
            for(col = 0;col < BOTTOM - MIDDLE;col++){
                memcpy(&frame2[IMAGE_WIDTH * IMAGE_DEPTH
                * (TOP + col) + WIDTH * IMAGE_DEPTH * index],
                &cell_image1[col * WIDTH * IMAGE_DEPTH], WIDTH
                * IMAGE_DEPTH);
            }
        }
        else{
            for(col = 0;col < BOTTOM - MIDDLE;col++){
                memcpy(&frame2[IMAGE_WIDTH * IMAGE_DEPTH
                * (MIDDLE + col) + WIDTH * IMAGE_DEPTH * index],
                &cell_image1[col * WIDTH * IMAGE_DEPTH], WIDTH
                * IMAGE_DEPTH);
            }
        }
    }
    xcommit(gid_me(gid), NUMBER_OF_WORKERS(int), TOP(int),
    MIDDLE(int), BOTTOM(int), WIDTH(int), --next_transaction(int));

    for(index = 0;index < IMAGE_WIDTH;index++){
        frame2[index + IMAGE_WIDTH * TOP] = (char)0xFF;
        frame2[index + IMAGE_WIDTH * MIDDLE] = (char)0xFF;
        frame2[index + IMAGE_WIDTH * BOTTOM] = (char)0xFF;
    }
    for(x = 0;x < IMAGE_WIDTH;x += WIDTH){
        for(y = TOP + 1;y < BOTTOM;y++){
            frame2[x + y * IMAGE_WIDTH] = (char)0xFF;
        }
    }
    for(index = 0;index < IMAGE_WIDTH * IMAGE_HEIGHT
    * IMAGE_DEPTH;index++){
        image[index * 4] = frame2[index];
        image[index * 4 + 1] = frame2[index];
        image[index * 4 + 2] = frame2[index];
        image[index * 4 + 3] = 0x00;
    }
```

```
            ximg = XGetImage(display, window, 0, 0, IMAGE_WIDTH,
                    IMAGE_HEIGHT, AllPlanes, ZPixmap);
            free(ximg -> data);
            ximg -> data = image;
            XPutImage(display, window, gc, ximg, 0, 0, 0, 0, IMAGE_WIDTH,
            IMAGE_HEIGHT);
        }
        gettimeofday(&last, NULL);
        printf("%ld\n", last.tv_usec - first.tv_usec);
    }
    free(frame1);
    free(frame2);
    free(image);
    free(cell_image1);
    free(cell_image2);
    XCloseDisplay(display);
    return 0;
}
```

2. Worker Program(clientmain.plc)

```
#include <stdio.h>
#include <plinda.h>
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>

#include "movement.h"

#define NAME_LENGTH  100

static gid gid_me;

static int IMAGE_WIDTH, IMAGE_HEIGHT;

int real_main(int argc, char *argv[], char *env[]) {
    char *image1, *image2;
    char name[NAME_LENGTH];
    char *result;
    int index;

    pl_arg_rdp(?name(char): 100, ?gid_me(gid));

    pl_rd[gid_me]("CONFIG", ?IMAGE_WIDTH(int), ?IMAGE_HEIGHT(int));

    image1 = (char *)malloc(IMAGE_WIDTH * IMAGE_HEIGHT);
    image2 = (char *)malloc(IMAGE_WIDTH * IMAGE_HEIGHT);
    init_movement(IMAGE_WIDTH, IMAGE_HEIGHT);
    while(1){
        int cells;

        xstart;

        pl_in[gid_me]("JOB", ?cells(int), ?image1(char): IMAGE_WIDTH
        * IMAGE_HEIGHT, ?image2(char): IMAGE_WIDTH * IMAGE_HEIGHT);
        treat_movement((unsigned char *)image1, (unsigned char*)image2);
        result = (char *)get_edge_image();
        for(index = 0;index < IMAGE_WIDTH * IMAGE_HEIGHT;index++){
            if(result[index] == (char)0xFF)
                image2[index] = (char)0xFF;
        }

        pl_out[gid_me]("RESULT", cells(int), image2(char): IMAGE_WIDTH
```

```
            * IMAGE_HEIGHT);
            xcommit();
        }
        close_movement();
        free(image1);
        free(image2);
        return 0;
}
```

ABSTRACT

Implementation of the Movement Detection Systems Using PLinda and Embedded System and Performance Analysis of Them

Hong, Seok Yong
Major in Computer System Engineering
Dept. of Computer System Engineering
Graduate School, Hansung University

One of the applications using CCD-cameras is the manless guard system which is likely to be used frequently in the world. It is monotonous and bothering that men always watch the images through the cameras. Moreover, in case serious guard is needed, intelligent camera systems can replace men as guards effectively.

Processing a lot of images from cameras needs no small memory space and time. And if that system works on embedded systems, there are some restrictions. So, in this research, an effective image-processing method is suggested and implemented for increase of the performance. And to use available resources in organizations, we make the movement detection program parallel using PLinda, one of the parallel systems, and estimate the suggested systems.